ChatGPT

# MarketScout SEO Supertool – Project Status and Next Steps

## 1. Project Documents & Architecture Summary

The **MarketScout SEO Supertool** is a modular web application built with a Node.js/Express backend and a React/Tailwind frontend. It integrates OpenAI's GPT-4 for intelligent SEO analysis, along with external APIs (SerpAPI for Google Maps data, etc.) to scout local business niches. The project's uploaded documents (roadmap, style guide, implementation guide, wireframes, and mockups) outline a clear vision:

- **UI/UX Design:** The Style Guide defines a clean, modern interface with the **Inter** font, a primary indigo accent color (`#6366F1` indigo-500) and consistent Tailwind utility classes for spacing, borders, and shadows. All components (buttons, cards, tables, form inputs) follow this cohesive design system. For example, primary buttons use indigo background and rounded-2xl corners, and form inputs use subtle gray borders and focus highlights. This ensures a unified look and feel across the app.

- **Frontend Structure:** The React frontend is organized into pages for each major tool: **Summary**, **Matrix**, **Heatmap**, **Scraper**, **Rank & Rent Bot**, **Estimator**, and **Memory**. A common `Layout` with a sidebar navigation hosts these pages. Each page is responsible for a specific feature:

- *Summary:* Overview tables of opportunity scores and full results.
- *Matrix:* An interactive filterable grid of opportunities (with sorting and favoriting).
- *Heatmap:* A map visualization plotting competitor locations color-coded by Opportunity Score.
- *Scraper:* CSV upload interface to initiate bulk data scraping.
- *Rank & Rent Bot:* A GPT-4 powered chat assistant for ad-hoc questions, which triggers targeted data scraping + analysis on the fly.
- *Estimator:* A lead value calculator for user-specified traffic and conversion assumptions (client-side only).

- *Memory:* (Planned) archive of saved chat answers or user-uploaded knowledge, not fully implemented yet.

- **Backend Structure:** The Node.js backend exposes RESTful APIs under `/api/*` for the above features. Notable endpoints include:

- `POST /api/run` – Bulk scrape runner for an uploaded CSV of keywords/locations. It uses **Multer** to accept files and SerpAPI to fetch Google Maps results for each keyword, then computes opportunity scores.
- `POST /api/bot` – The chat bot endpoint. It accepts a natural language question, uses GPT-4 function calling to extract the target location and niches from the query, then performs a focused

scrape + scoring for those niches. Results are returned in a structured JSON (including a summary and CSV link) for the frontend to display.
- `GET /api/summary` – Returns the latest **Opportunity Summary** (a compiled CSV of competitor audit results and scores) as JSON. There are also sub-routes to download raw CSV files (e.g. full results, summary).
- `GET /api/matrix` – (If applicable) Possibly similar to summary or deprecated (the Matrix page currently reuses summary data).
- `GET /api/heatmap` – Provides latitude/longitude and scores for plotting on the map.
- `POST /api/ingest` – (Recently added) Allows uploading documents to enrich the AI's knowledge base. This hints at a LangChain/FAISS integration for vectorized document retrieval, though the implementation details are forthcoming.

These modules interact via a **modular architecture** – the frontend calls backend endpoints (running on port 3001) for heavy tasks, while keeping UI state and interactivity client-side. OpenAI is used on the backend via the official API (OpenAI Node SDK) for both chat completions and embeddings. The project plans mention using **LangChain** and **FAISS**: these will likely support advanced AI features such as semantic search over content or caching embeddings for faster responses. In the current code, we see groundwork for this: e.g. a simple vector store (JSON files `chunks.json` / `embeddings.json`) is used in an older `/ask` endpoint for retrieval-augmented answers. Going forward, this will be replaced with a more robust LangChain + FAISS solution for storing and querying embeddings (improving the AI assistant's ability to use reference documents).

**Caching & Batching:** The roadmap emphasizes efficient API usage. Indeed, we must batch operations where possible and avoid unnecessary calls. The design already groups keyword searches by location in the scraper (processing one location's keywords together) and deduplicates results by Place ID. We should extend similar care to OpenAI calls (e.g. consolidating multiple content generations into one request when building pages, as discussed below). Caching strategies (like storing results of a scrape or reuse of embeddings) will prevent redundant work. The current codebase uses local storage for chat memory caching on the frontend and writes output files (CSV/JSON) on the backend that persist results. These practices align with the plan's guidance to **"respect caching, batching, and API usage limits."**

In summary, the project's foundation is strong. All core analysis features (data scraping, competitive audit scoring, interactive visualization, Q&A assistant) are in place per the roadmap's initial phases. The UI/UX is guided by a polished style guide ensuring consistency. The next step is to build on this foundation without regressing any existing functionality – especially the patched modules (`Bot.jsx`, `ask.js`, `run.js`, `bot.js`, etc.) that are now stable. New development should integrate **seamlessly** into the current structure, following the established patterns.

## 2. Next Urgent Development Task from the Roadmap

According to the roadmap and project plan, the **next major milestone** is to implement the **"Site Builder"** feature. This is clearly indicated by the provided wireframes and mockups for a *Site Builder UI* (see the design mockup below) and is the logical next step after identifying promising niches. The Site Builder will allow users to **generate a starter website or landing page** for a chosen niche/location, using AI to draft content. This moves the project from analysis into execution – fulfilling the "Supertool" promise of not only scouting opportunities but also acting on them.

*Polished mockup of the Site Builder UI. The design shows form fields for location, niche, competitor references, and AI-generated outputs like SEO title, meta description, headings, etc., with options to export or deploy the site.*

Several reasons make Site Builder the top priority now:

- **Roadmap Position:** The roadmap's timeline (2025) places content creation tools right after the data analysis phase. Core analysis (Matrix, Heatmap, Bot) is done, so the team is ready to tackle site generation as the next big feature.
- **Design Artifacts:** The presence of high-fidelity UI designs for Site Builder indicates it has been planned and scoped in detail. It's a key feature waiting to be built, unlike smaller enhancements (e.g. settings panel) which are important but likely scheduled slightly later.
- **User Value:** From a user's perspective, after identifying a high-opportunity niche with the tool, the natural next step is to quickly build a lead-gen site for that niche. Delivering this feature will significantly increase the product's value – turning insights into action.
- **Integration of AI & Tools:** The Site Builder will heavily leverage the OpenAI integration (for content generation) and possibly LangChain tool chains (for pulling in competitor info or structuring content). This aligns with the project's emphasis on using AI and making the most of the OpenAI/LangChain stack we've set up. It's a showcase of these technologies working together.
- **Completeness of Current Features:** Most other planned components are either complete or in maintenance mode. For instance, **Search Volume & CPC integration** (to enrich the Opportunity scores) is one pending item, as evidenced by the "Volume" and "CPC" columns in the Summary UI that are not yet populated. However, incorporating that data is likely a smaller task that can be handled alongside or just before Site Builder. The large architectural piece that remains is Site Builder itself. (We will still ensure to factor volume/CPC data into the Opportunity Score formula when refining the scoring algorithm, but this is part of ongoing enrichment rather than a full new module.)

In summary, **Site Builder is the next urgent development task** per the roadmap. It represents the next phase ("Execution Tools") after the current analysis toolkit. We will proceed to outline a detailed plan for implementing the Site Builder, making sure it adheres to the provided UI/UX design and the project's architectural standards.

# 3. Build Plan: Implementing the Site Builder Module

Developing the Site Builder will be a significant feature addition. We will approach it in structured steps, ensuring alignment with the design mockups and maintaining clean integration with the existing codebase. Below is a detailed plan:

### 3.1 Frontend – Site Builder Page UI

- **Create a New React Page:** We will add a new page component `SiteBuilder.jsx` under `frontend/src/pages`. This page will be added to the router and navigation (similar to how existing pages like `Scraper` or `Estimator` are set up in `Layout.jsx`). The nav label will be " Site Builder" with route `/site-builder` (the exact emoji/icon can match design, e.g. a drafting compass as in the mockup).
- **Form Layout:** Implement the form UI exactly as per the mockup【80†】. This includes:

- Two primary text inputs for **City** and **Niche**. These will capture the target location and service for the site. We'll reuse styling from the Style Guide for inputs (e.g. the standard input class with `border-gray-300 rounded-md px-3 py-2 text-sm focus:ring-indigo-500` for focus). Labels should be clear (e.g. "City" and "Niche or Service Type").
- A section for **Competitor Reference Content**. This might be a large text textarea or a list input. According to the mockup, it shows "Competitor Reference – Top sites" with options to *Prefill* or *Import from AI*. For MVP, we can have a multiline textarea where the user can paste content or notes from top competitor sites. We'll also include a helper text or a button to fetch competitor sites:
    - *Prefill:* We can populate this textarea with something like the top 1–3 competitor website URLs or names, drawn from the existing scraped results. For instance, since we already have `Website` URLs from the Maps results (in `results.csv` and combined matrix data), if the user has run a scrape or bot query for this niche/location, we could show those top competitor URLs here. This provides an easy starting point.
    - *Import from AI:* As an advanced option, we could allow the app to automatically fetch and summarize competitor site content. This would use a background process (possibly an OpenAI browsing plugin or a server-side scraper). However, this can be complex to implement fully. In the first iteration, we might stub this out or simply guide the user to paste content. (We'll note to implement automated competitor content import in a later iteration using LangChain's web scraping tools or an API.)
- Fields for **SEO Title** and **Meta Description** – these will be output fields that the AI generates. In the UI, these can be disabled text inputs or textareas that get filled after generation. We should style them as read-only fields initially (or editable if we allow user tweaking).
- Fields for **Main Heading** (H1) and possibly sub-sections (the mockup shows "Services Section") – these represent the main content blocks of the landing page. We'll likely treat these as textareas or a small rich-text editor where AI output appears and can be edited. For simplicity, use textareas styled per the guide (e.g. class `rounded-md border px-3 py-2 shadow-sm w-full` for a textarea).
- An input for **Phone Number** – presumably to include a call-to-action phone contact on the landing page. This can be a simple text input (with basic validation for numeric characters).
- **Hosting Mode Options:** The mockup shows radio buttons for "Free Temporary" vs "Full Hosted". For now, we will include these as radio inputs in the form, but their functionality can be limited at first:
    - "Free Temporary" could mean the site is just downloadable or served locally.
    - "Full Hosted" might imply deploying to a live server or service (which likely requires integration with a hosting provider or an S3 bucket, etc.). Implementing full hosting is complex and may be planned for a later stage; we will design the UI for it but possibly grey it out or display a "Coming Soon" note for the initial version.
- **Generate & Output:** At the bottom, there will be two main action buttons:
- **"Generate Site Content"** (this is implied by the mockup – possibly when the user toggles something or on form submission). We will include a primary button (styled as per guide: indigo background, white text, etc.) that says "⚙ Generate Content" or simply "Generate Site". When clicked, this will trigger the content generation process (calling the backend API, see below). While generation is running, we should provide feedback (e.g. a loading spinner or "Generating..." text on the button, disabling it to prevent duplicate calls).
- **"Export HTML"** – after generation, allow the user to download the generated site. This button will likely become enabled once content is ready. It will bundle the filled content into an HTML template and trigger a download. We can implement this by having the backend provide the HTML (or by constructing it on the client). A simple approach is to have a route like `/api/site/export` that returns the HTML string with appropriate headers so the browser downloads it as `.html`.

- **"Deploy Now"** – also shown in the design. For MVP, we might implement a minimal deployment (for example, hosting the page at a known URL on the Node server or using a service like Netlify via API). However, given time constraints, this could be stubbed initially. We might have it open a modal saying "Deployment feature coming soon" or only log the action. If we do implement it, one quick solution is to use the Express server to serve the generated HTML at a route (e.g. `/site_preview`) so the user can at least view it in-browser. A full integration to a hosting provider can be planned for a later sprint.

- **Ensure Responsive and Clean UI:** We'll follow the style guide for spacing and make sure the form wraps nicely on smaller screens (Tailwind's grid and flex utilities can be used as in the mockups). Each section of the form (inputs, outputs, etc.) should be contained in card-like panels or well-separated with headings, matching the grayscale wireframes and polished UI. For example, use a white card container with shadow for the output section to distinguish it (the style guide suggests using `rounded-2xl bg-white shadow-md p-6 flex flex-col gap-2` for cards).

- **Integration in App:** Add the new route to the React Router (likely in `App.jsx` or wherever routes are defined). Also include the new nav item in the sidebar (with proper icon and active styling consistent with others – active tab gets indigo highlight per style guide).

### 3.2 Backend – Content Generation API

- **New Express Route:** Implement a new backend route, e.g. `backend/api/site.js`, mounted at `/api/site` (or `/api/generateSite`). This will handle POST requests from the Site Builder form. We'll register it in `server.js` just like other feature routes. The route will:
- Accept a JSON payload containing the user inputs: location (city), niche (service type), possibly competitor URLs or content (if the user provided any), phone number, and hosting preference.
- Upon receiving a request, validate the inputs (using simple checks or a schema, possibly with Zod or similar since the project already includes Zod in dependencies).
- **AI Content Generation:** This is the core of the backend work. We will use the OpenAI API to generate the site content. To do so while respecting API limits and ensuring quality:
    - **Prompt Engineering:** Construct a single prompt that instructs GPT-4 to produce all necessary content for the landing page in a structured format. For example, we can prompt something like: *"You are an AI copywriter specialized in local business websites. Generate a landing page for a [niche] in [city]. Include: an engaging SEO title (max 60 characters), a meta description (160 characters), a catchy main heading, a brief intro or services section (~2-3 sentences), and a call-to-action that includes phone [phone number]. Use a friendly, professional tone. Output the result in JSON with keys: seo_title, meta_description, heading, intro_section."* We will tailor this prompt based on the style we want. The style guide doesn't explicitly cover writing tone, but we can assume a professional, concise style is desired.
    - **Incorporating Competitor References:** If the user provided competitor site content or URLs, we should leverage that to make the generated content more targeted. There are two approaches:
    - *Simple approach:* Append a summary of competitor offerings to the prompt. For example, if the user pasted some notes like "Competitor A emphasizes 24/7 service, Competitor B has no mention of warranty", we include: *"Competitor insights: [...]. Based on these, differentiate our content."* This can guide the AI to produce unique selling points.

- *Advanced approach:* Use LangChain to fetch and summarize competitor websites. We could programmatically retrieve the HTML of the provided URLs, use a library or LangChain's HTML loader to extract text, and then either summarize it or convert it into vectors. Given the time, a summarized inclusion might be easiest: for each competitor site, use OpenAI in a short call to get a 1-paragraph summary of key points (ensuring we don't exceed context limit). Then feed those summaries into the main prompt as context.
- If the content is large, a LangChain+FAISS pipeline could be used: embed competitor content into a vector store and do a similarity search for relevant points about services or offers, then include top matches as context. However, this might be overkill for a single landing page generation, and the prompt length might suffice without full vector search. Regardless, this is an opportunity to apply LangChain if needed – so we will design the code to potentially use a vector store if multiple competitor docs are involved (the infrastructure for FAISS is planned, per the roadmap).
- **OpenAI Function Calling:** To get structured output easily, we can use the function calling feature of the OpenAI API (as was done in the Bot module for extracting niches). Define a function schema like `generate_site_content` with parameters: `seo_title`, `meta_description`, `heading`, `intro_section` (and any other sections we want, possibly `services_points` if needed as a list). The model can then return a JSON object neatly filled. This saves us from having to parse text for each field. We have a working example of function calls in the codebase to reference, ensuring consistency in approach.
- **API Call:** Use the OpenAI GPT-4 model (already configured in the project via the OpenAI SDK) to create the completion. We will set up the messages array with a system prompt that includes instructions and possibly the style guide pointers (to maintain tone), followed by a user prompt that contains the specifics (city, niche, competitor info). Then attach the function definition for `generate_site_content`. We expect the model to return a function_call result with arguments for our content.
- **Error Handling & Token Limits:** We should guard against too-large inputs (if a user pastes very large competitor text, we might need to truncate or summarize first to keep within GPT-4 token limit ~8K or 16K). We'll also handle API errors (network issues, etc.) and return a user-friendly error message JSON if generation fails.

- **Construct HTML Template:** Once we have the generated content (title, meta, heading, body text, etc.), the backend can also assemble a simple HTML page string. We can have a basic HTML template file (with inline CSS for simplicity or use Tailwind styles via CDN) where we inject the generated text. For example, an `<html>` with `<head><title>seo_title</title><meta name="description" content="...">...</head>` and `<body>` containing an `<h1>` for heading, `<p>` for intro, maybe a `<section>` for services if provided, and a `<footer>` with contact info (phone number). This template ensures the export and deploy have a consistent structure.

- **Response:** The `/api/site` POST should respond with a JSON indicating success and the generated content. For instance, `{ success: true, content: { seo_title: "...", meta_description: "...", heading: "...", intro_section: "..." } }`. We might not send the entire HTML here (to avoid escaping issues in JSON), but we will save it server-side for download. Alternatively, we could respond with a unique ID or filename where the HTML was saved.

  - We will also consider caching the result: if the same niche & city were requested recently, we could reuse the last generated content to save tokens. This might be an optimization; initially,

we can implement without caching, but design the code so that caching can be added (e.g. save generated content keyed by niche+city to a JSON file or database).

- **Export & Deploy Endpoints:** Alongside the main generation:

- Implement `GET /api/site/download` (or extend the summary download route) to serve the generated HTML file. For example, if we saved the file as `output_site.html` on the server (or a temp file), this endpoint can send it with `res.download()` so the user's browser downloads a ready-to-use HTML file (similar to how summary CSV download is handled).
- Implement (optionally) `POST /api/site/deploy` for the "Deploy Now" action. In this function, we could integrate a third-party service or simply host the file on our server:
    - A simple approach: Serve the file statically from the `/data` or a `public/sites` directory and provide the user a URL (like `http://localhost:3001/sites/landing_<city>_<niche>.html`). This requires copying the file to a static dir and ensuring Express can serve it (we might add an Express static middleware for a `/sites` folder).
    - We will return that URL in the response so the frontend can inform the user ("Your site is live at ..."). For production deployment in the future, a more robust solution (Netlify, S3+CloudFront, etc.) can replace this, but our architecture should allow swapping the deployment mechanism easily (perhaps abstract in a deployService module).
- **Avoid Breaking Existing Modules:** It's crucial that adding Site Builder doesn't interfere with current features:

- We will **not modify** core flows of `run.js` or `bot.js` except maybe trivial things (like adding volume data if needed). The new code will be mostly self-contained in the new route and page.
- Ensure the `combined_opportunity_matrix.csv` and other data files remain untouched by the site builder process (except reading them for competitor info). Site generation should be side-effect free on those files.
- The only integration point might be if we choose to auto-prefill competitor references from recent results; we'll then only **read** from the results CSV or summary JSON (via `/api/summary`) instead of doing new scrapes.
- We must also respect API usage: calling OpenAI for site content is an additive usage. We might implement basic rate limiting (e.g. one generation at a time per user) if necessary, and log usage for monitoring. The code should check for an `OPENAI_API_KEY` as it does now and respond gracefully if not set.

**3.3 Incorporating Keyword Volume & CPC (Enrichment)**

*(Note: This is a parallel enhancement, not the primary task, but it ties into finalizing the Opportunity Score and could be addressed during Site Builder implementation.)*

Before or during the Site Builder work, we should finalize the **Opportunity Score** calculation using search volume and CPC data. The Summary table in the UI has placeholders for "Volume" and "CPC", and the Opportunity Score/Level should factor these in for a truly meaningful ranking.

- We will integrate the Google Ads Keyword Planner API (or an alternative source) to fetch monthly search volume and average CPC for each keyword (niche + location) that the user is investigating. The original plan (Group2 in baseline) included scripts for Google Ads; we can use that approach:
- After scraping competitors in `run` or `bot`, take the list of unique keywords and query the Google Ads API in batch for their stats. This can be done server-side, likely in the `auditAndScore` phase or just before it.
- Store the Volume and CPC alongside each keyword's data (perhaps extend the `summaryData` objects with these fields).
- Update the `auditAndScore` logic to compute an **OpportunityScore** that combines factors:
    - e.g. Start with a base score derived from Volume (higher volume = higher base score) and CPC (higher CPC = higher base). Then *adjust* based on competitive factors: e.g. +points if Map Pack count < 3 (thin competition), +points if any low-review or missing-site (weak competition), –points if many competitors have high reviews. We will devise a formula and document it in code comments for transparency.
    - Also categorize **OpportunityLevel** as High/Medium/Low based on score thresholds (e.g. top 25% scores = High, etc.), which the UI already expects and styles.

- This change will reflect in the Summary and Matrix automatically, populating the Volume/CPC columns and refining the sort order of opportunities. We'll verify that the front-end Summary.jsx is accessing the right field names (the UI expects `row.Volume` and `row.CPC` and `row.Opportunity_Score` with underscore or space – we might adjust keys to match, e.g. use `Opportunity Score` with space in CSV but convert to `Opportunity_Score` in JSON). Once implemented, this enrichment makes the **analysis phase feature-complete**, and it complements the Site Builder by giving users more confidence in the niches they choose to build a site for.

- We must ensure this volume/CPC lookup is **cached** effectively: perhaps maintain a local JSON of keyword -> volume/CPC to avoid querying the API repeatedly for the same terms. Given API limits, this is important. We can update the `services_list.csv` or have a new `keyword_stats.json` for caching these values.

*(Volume/CPC integration is a secondary task here, but it's highlighted to ensure completeness. If timing is tight, the Site Builder can be implemented first using the current Opportunity Score, and the enrichment can follow shortly after, as it won't heavily affect the Site Builder code.)*

**3.4 Testing and QA**

- With the Site Builder frontend and backend in place, we will thoroughly test the end-to-end flow:
- Enter various inputs (different niches and cities) and generate content. Check that all fields get sensible output and the JSON parsing works. We'll verify the structure of the HTML export (open the downloaded HTML in a browser to ensure it's well-formed and styled).
- Test with and without competitor reference text to see how that influences output. If possible, test the automated competitor "Prefill" to ensure it grabs valid data (like top 3 competitor names or URLs from the last run).

- Ensure that generating a site without first running a scrape is still possible (the user might directly use Site Builder). In that case, no prefill data – the AI should still generate generic content. That's fine.
- Check that none of the existing features are broken: run a scrape, open matrix, ask the bot a question – all should still work as before. The addition of the new route should not interfere with, for example, the Bot's function calling or the summary generation.

- Test the concurrent usage: trigger a site generation while a scrape might be running, etc. The server should handle it (perhaps queue internally if needed). Given Node's single-threaded nature for JS, heavy parallel calls might slow down, but our use of async/await will let Node handle interleaving I/O. It should be okay, but we'll keep an eye on performance and maybe introduce job queueing in the future if needed.

- **UI Polish:** Compare the implemented UI to the provided high-res mockups and the style guide. Fine-tune padding, font sizes, and iconography to match. For instance, headings should use the designated font sizes (H2 for section titles like "SEO Title" label should probably be `text-xl font-semibold` per style guide). Ensure all text uses the correct weights and colors (e.g. placeholder or helper text in forms could be gray-500). Use Lucide or Heroicons for any new icons (the mockup shows icons next to field labels like a link icon for competitor sites – we can include those from Lucide as needed, consistent with how other pages use icons like Stars in Matrix).

By following this plan, the Site Builder feature will be implemented in a way that **complements the existing toolset**. Users will be able to seamlessly go from identifying an opportunity to crafting a basic site for it, all within MarketScout. Crucially, we adhere to the established conventions: API usage is done server-side with careful prompt design (leveraging function calls similar to earlier features), the UI matches the approved design, and no existing module is disrupted by our additions.

## 4. Git Workflow & Branch Strategy

To implement these changes, we will use a structured Git workflow, creating a feature branch and making incremental commits:

1. **Create a Feature Branch:** Following the naming scheme, we'll create a branch called `MarketScout - Site Builder` (including the feature name in the branch). All development for the site builder will happen on this branch.
2. **Incremental Commits:** We will commit logically separated chunks of work with clear messages. For example:
3. *"feat(site-builder): add SiteBuilder page component and route"* – committing the initial frontend page and nav integration.
4. *"feat(site-builder): implement Site generation API endpoint with OpenAI integration"* – committing the new backend route and OpenAI logic.
5. *"feat(site-builder): add volume & CPC enrichment to scoring (keyword stats integration)"* – if we include the volume/CPC in this sprint, commit that separately.
6. *"style(site-builder): apply style guide styles to Site Builder form and outputs"* – fine-tuning the CSS and UI details.

7. *"chore: update summary download routes to include site export"* – committing any ancillary changes like download endpoint or minor refactors. Each commit will reference the feature context and keep changes focused, making it easier to review.
8. **Testing on Feature Branch:** Push the branch to the repository and have the team (or CI) test it. Ensure code review is done given this is a large addition. Resolve any conflicts with the main branch (for instance, if other small patches went into main, rebase or merge as needed).
9. **Pull Request:** Open a PR titled "MarketScout – Add Site Builder Feature" describing the changes. In the PR description, outline how the feature works and reference the relevant parts of the roadmap ("Implements Site Builder as per Q2 2025 roadmap item") so stakeholders can connect it with plan expectations.
10. **Review & Merge:** After approvals, merge the feature branch into the main branch (or develop branch if using Git Flow). Use a merge commit that perhaps squashes minor commits if needed, but preserving the history is fine since our commits are well-organized.
11. **Tagging:** Optionally, tag this release as a new version (e.g. `v0.2.0` if the initial analysis toolkit was v0.1) to mark the addition of a major feature.
12. **Post-Merge Checklist:** After merging, verify that all environment keys (OpenAI API key, SerpAPI key, Google Ads credentials if added) are configured in production as needed. Given we added possibly new config (Google Ads), update the `.env.example` or README accordingly in a documentation commit.

Throughout development, we avoid touching the patched files from earlier hotfixes except when absolutely necessary. For instance, if we adjusted `auditAndScore` for volume integration, that's within scope – but we won't alter the logic of `Bot.jsx` or `ask.js` or others that were recently patched for stability. This isolated branch strategy guarantees we don't "break" existing functionalities; if any issues arise, we can easily compare against the main branch. The commit history will also make it straightforward to roll back specific changes if something inadvertently affected an existing module.

By following this Git workflow and branch naming convention, we ensure that our **"MarketScout - Site Builder"** feature is developed in a controlled, traceable manner. This makes code reviews efficient and maintains the project's professional development standards.

---

**Sources:**

- MarketScout SEO Style Guide (design system and component styles)
- MarketScout UI Design Mockups (Site Builder)【80†】
- MarketScout Codebase (Git repo files): Express backend and React frontend structure, Bot/AI integration examples, and Summary/Matrix UIs.

---